A commutative replicated data type for cooperative editing

Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, Mihai Leția

2009 29th IEEE International Conference on Distributed Computing Systems

A commutative replicated data type for cooperative editing

Nuno Preguiça, CITI/FCT-Universidade Nova de Lisboa
Joan Manuel Marquès, Universitat Oberta de Catalunya
Marc Shapiro, INRIA Paris-Rocquencourt and LIP6
Mihai Leția, École Normale Suprieure de Lyon and LIP6

Abstract

A Commutative Replicated Data Type (CRDT) is one where all concurrent operations commute. The replicas of a CRDT converge automatically, without complex concurrency control. This paper describes Treedoc, a novel CRDT design for cooperative text editing. An essential property is that the identifiers of Treedoc atoms are selected from a dense space. We discuss practical alternatives for implementing the identifier space based on an extended binary tree. We also discuss storage alternatives for data and meta-data, and mechanisms for compacting the tree. In the best case, Treedoc incurs no overhead with respect to a linear text buffer. We validate the results with traces from existing edit histories.

1. Introduction

To share information, users located at several sites may independently update a common object, e.g., a text document. Each user operates on a separate *replica* (i.e., local copy) of the document. A well-studied example is co-operatively editing a shared text.

As users make local modifications, replicas diverge from one another. Operations *initiated* on some site propagate to other sites and are integrated or *replayed* there. Eventually, every site executes every action.

Despite this, replicas might not converge if they execute operations in different orders. In order to guarantee convergence, two basic approaches can be found in the literature. First, serializing, i.e., enforcing a total order of operations before execution [1]. The extra serialization delay is usually unsuitable for cooperative editing systems. Second, operational transformation, i.e., modifying the parameters of operations to make them run in different orders [2]. This approach is complex and error-prone, as evidenced by the errors found in published algorithms [3].

We suggest a different approach: to *design* replicated data types such that concurrent operations *commute* with one another. Let us call such a type a *commutative replicated data type* or CRDT. If operations replay in happened-before order, replicas of a CRDT converge automatically, without complex concurrency control.[1] The interested reader will find a proof of this property in our technical report [4].

Hereafter, we study a CRDT for cooperative editing, i.e., a shared sequential buffer. Each buffer entry is identified by a unique identifier. To provide the CRDT property requires that identifiers be unique, stable, ordered (in the same order as the buffer), and *dense*. This means that it is always possible to create a new identifier between two existing ones. The challenge studied in this paper is to keep identifiers short and to minimize overhead.

Internally, we use an extended binary tree for identification and storage, hence the name *Treedoc*. We study two identification alternatives: one is compact but uses tombstones to keep track of deleted entries; the other allows deleted entries to be discarded immediately. As tree unbalance causes overhead, we suggest optimizations to avoid unbalance.

Furthermore, we propose a compaction mechanism that removes storage overhead and shortens the identifiers within "cold" regions of the document. In the best case, a compacted Treedoc reduces to a sequential array, with zero overhead.

We validate our design with a benchmarking study, based on traces from existing editing histories. We conclude that the costs of Treedoc are reasonable, and better than competing CRDTs.

The contributions of this paper are the following:

- We propose the CRDT method for designing shared replicated data types, ensuring convergence without complex concurrency control. We illustrate the CDRT method with a useful, concrete example: a shared sequential edit buffer. We identify the requirements for the edit-buffer, in particular, a dense identifier space.
- We propose a practical and efficient implementation of a dense identifier space, based on an extended binary tree.
- We propose a number of optimizations that compound identifier and storage overhead.
- We propose a distributed compaction mechanism that minimizes identifier size and removes storage overhead.

1. Our *happened-before* and *concurrency* relations are identical to the formal definition of Lamport [1].

1063-6927/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDCS.2009.20

The remainder of this paper proceeds as follows. Section 2 describes the generic shared buffer data type. Section 3 present Treedoc, a practical implementation of the previous abstraction. Optimizations to the basic design are presented in Section 4. Section 5 presents an evaluation of Treedoc. Section 6 compares with previous work. Section 7 concludes.

2. Requirements

In this section we consider the requirements and interface of a CRDT data type for concurrent editing, independently of implementation. Consider a shared, replicated document, consisting of a linear sequence of elements called *atoms*. An atom may be a character or some other non-editable element, e.g., a graphics file inserted inside the document [5].

2.1. Unique position identifiers

Each atom has an associated unique position identifier (PosID), with the following properties: *(i)* Each atom in the buffer has an identifier. *(ii)* No two different atoms have equal identifiers. *(iii)* The identifier of a given atom remains constant for the whole lifetime of the document. *(iv)* There is a total order of identifiers, noted $<$, consistent with the order of atoms in the buffer. *(v)* The identifier space is *dense*: given any identifiers P and F, an identifier between P and F can be found. Formally: $\forall P, F : P < F \Rightarrow \exists N : P < N < F$.

Later in this paper, we will relax requirements *ii* to allow recycling unused identifiers, and *iii* to allow non-ambiguous renaming.

A dense identifier space is infinite. We define an identifier U to be *used* if the document contains an atom identified by U, and unused otherwise.

To insert an atom between positions P and F requires allocating a *fresh* identifier N, i.e., a previously-unused one, with $P < N < F$. The density property ensures this is always possible.

Rational or real numbers are dense, but they would require infinite precision, which is not practical. In Section 3 we will present a practical alternative, based on binary trees.

2.2. State and Operations

We define an abstract atom buffer data type whose state T is a set of $(atom, PosID)$ couples, where $PosID$s are unique. The contents of state T is the sequence of all $atom$s in T ordered by $PosID$.

Each user can maintain a replica of the data type and modify it locally by initiating one of the following *edit operations*:

- $insert(PosID_n, newatom)$, where $PosID_n$ is a fresh and unique identifier, inserts atom $newatom$ into the document state.
- $delete(PosID_n)$ removes the atom with PosID $PosID_n$ from the document state. In the initiator's state, there must be an atom with PosID $PosID_n$.

Assuming PosIDs satisfy the requirements of Section 2.1, this design ensures that any pair of concurrent operations commute.

Indeed, two operations that refer to different PosIDs are independent. As their effect on the data type is independent of execution order, they commute. The requirement that $PosID_n$ be unique ensures that two concurrent inserts commute. Consider now concurrent operations that refer to the same unique identifier. An insert must happen-before a delete with the same identifier, they can never be concurrent. Finally, the delete operation is idempotent; therefore concurrent deletes of the same PosID have the same effect in whatever order. This shows that the abstract buffer is a CRDT.

Multiple users may edit a document concurrently. Updates received from remote sites may be replayed as soon as received, as long as happened-before order is satisfied. Since the abstract document type described here is a CRDT, convergence is guaranteed even though operations execute in different orders at the different replicas.

3. Treedoc

In this section we present a practical implementation of the abstraction, using a tree structure for atom identifiers.

In the following examples, we take an atom to be a single character for illustrative purposes. (The evaluation of Section 4 uses whole lines for atoms.)

3.1. Paths

In order to satisfy the density requirement of Section 2.1 in an efficient way, the basic idea is to use *paths* in a *binary tree*. The total order of identifiers is given by walking the tree in infix order. For example, Figure 1 is one possible representation of a document containing the characters `"abcdef"`.

The figure shows data stored as nodes of the identification tree. This is only one possible implementation. Alternatively, storage may be decoupled from identification; for instance, as a set of of $(atom, PosID)$ pairs. In practice, an implementation can choose whichever is most efficient dynamically. Later in this paper we will show how to switch between tree and array storage in order to avoid overhead.

This binary tree identification structure is insufficient for concurrent edits, as users might concurrently insert two different atoms at the same position. To address this issue, we extend the nodes of basic binary tree, to contain any number of internal *mini-nodes*. A node containing mini-nodes will be called a major node, or just node when there

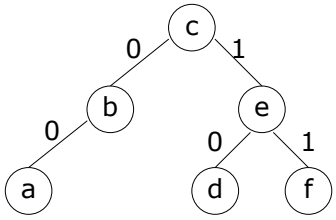

Figure 1. Identifiers in a shared text buffer, single-user version

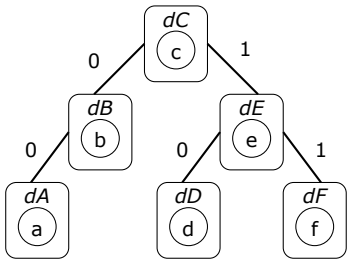

Figure 2. Identifiers in a shared text buffer, multi-user version of Figure 2

is no ambiguity. Figure 4 (node 100) illustrates a major node with mini-nodes.

Inside a major node, mini-nodes are identified by a *disambiguator*. Disambiguators must be unique and ordered. (Section3.3 discusses alternatives for generating disambiguators.)

Paths (PosIDs) include a disambiguator only when necessary, i.e., *(i)* at the last element of the path; or *(ii)* whenever the path follows a child of a mini-node explicitly. A path element without a disambiguator refers to the children of the corresponding major node.

The path to the root is the empty bitstring ϵ; the path concatenation operator is noted $\odot$. We note $[b_1 \dots b_n]$ for $b_1 \odot \dots \odot b_n$ when there is no ambiguity (we always omit ϵ when representing paths).

Figure 2 presents the example of Figure 1 with the extended tree structure. Here, user A with disambiguator dA has inserted atom a; user B with disambiguator dB inserted atom b; and so on.

The order on identifiers is defined by infix-order walk of the whole tree. A major node is ordered by infix-order walk: the major node's left child is before any mini-node; mini-nodes are ordered by disambiguator; and mini-nodes are before the major node's right child.

Formally, $id_1 < id_2$, iff:

- $id_1 = c_1 \odot \dots \odot c_n$ is a prefix of $id_2 = c_1 \odot \dots \odot c_n \odot j_1 \odot \dots \odot j_m$ and $j_1 = 1 \vee j_1 = (1 : d), \forall d$, or
- $id_2 = c_1 \odot \dots \odot c_n$ is a prefix of $id_1 = c_1 \odot \dots \odot c_n \odot i_1 \odot \dots \odot i_m$ and $i_1 = 0 \vee i_1 = (0 : d), \forall d$, or
- $id_1 = c_1 \odot \dots \odot c_n \odot i_1 \odot \dots \odot i_n$ has a common prefix with $id_2 = c_1 \odot \dots \odot c_n \odot j_1 \odot \dots \odot j_m$ and $i_1 < j_1$.

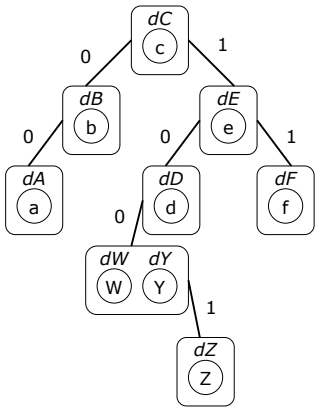

Figure 3. Identifiers after concurrent inserts

Given $i_1 = p_i \in \{0, 1\}$ and $j_1 = p_j \in \{0, 1\}$, we say that $i_1 < j_1$, iff: $p_i < p_j$.
Given $i_1 = (p_i : d_i)$ and $j_1 = (p_j : d_j)$, we say that $i_1 < j_1$, iff: $p_i < p_j \vee (p_i = p_j \wedge d_i < d_j)$.
Given $i_1 = 0$ and $j_1 = (p_j : d_j)$, we say that $i_1 < j_1$.
Given $i_1 = (p_i : d_i)$ and $j_1 = 1$, we say that $i_1 < j_1$.

We define the ancestry of a node. Node u is the (direct) parent of node v, noted u/v, iff $id(v) = id(u) \odot p \vee id(v) = id(u) \odot (p : d), \forall p, d$; equivalently, v is a (direct) child of u. Node u is an ancestor of v (or, equivalently, v is a descendant of u), noted $u/^+v$, if u is a parent of v, or recursively a parent of an ancestor of v (i.e., a grand-parent, or great-grand-parent, etc.).

Mini-node u is a *mini-sibling* of v, noted $MiniSibling(u, v)$, if they are mini-nodes of the same major node.

3.2. Generating fresh PosIDs for insert

When inserting a new atom, we will create a new PosID by adding a new node in the tree of unique identifiers. When inserting between mini-siblings of a major node, a direct descendant of the mini-node is created. Otherwise, a child of a major node is created.

Algorithm 1 presents a simple approach for generating a fresh new PosID for inserting between atoms with PosIDs $PosID_p < PosID_f$. Without loss of generality, we assume that in the shared buffer there is no used $PosID_m$ such that $PosID_p < PosID_m < PosID_f$. The algorithm allocates a fresh identifier, by creating a new child to the right of $PosID_p$ or to the left of $PosID_f$, depending on which has no descendant.

Let us consider again the example of Figure 2. When one user inserts `Y` between `c` and `d`, the PosID of `Y` will be $[10(0 : dY)]$ (using rule in line 4 - $PosID_c$ is an ancestor of $PosID_d$, thus $PosID_d$ has no left child). Thereafter, when inserting `Z` between `Y` and `d`, the new PosID will be $[100(1 : dZ)]$ (using rule in line 5 - $PosID_Y$ is a descendant of

Algorithm 1 New unique identifier for insert

1: **function** $newPosID\ (PosID_p, PosID_f)$
2: // d: new disambiguator.
3: **Require:** $PosID_p < PosID_f \wedge \nexists$ atom x such that $PosID_p < PosID_x < PosID_f$
4: **if** $PosID_p /^+ PosID_f$ **then** Let $PosID_f = c_1 \odot \ldots \odot (p_n : u_n)$; **return** $c_1 \odot \ldots \odot p_n \odot (0 : d)$
5: **else if** $PosID_f /^+ PosID_p$ **then** Let $PosID_p = c_1 \odot \ldots \odot (p_n : u_n)$; **return** $c_1 \odot \ldots \odot p_n \odot (1 : d)$
6: **else if** $MiniSibling(PosID_p, PosID_f) \vee \exists PosID_m > PosID_p : MiniSibling(PosID_p, PosID_m) \wedge PosID_m /^+ PosID_f$ **then return** $PosID_p \odot (1 : d)$
7: **else** Let $PosID_p = c_1 \odot \ldots \odot (p_n : u_n)$; **return** $c_1 \odot \ldots \odot p_n \odot (1 : d)$

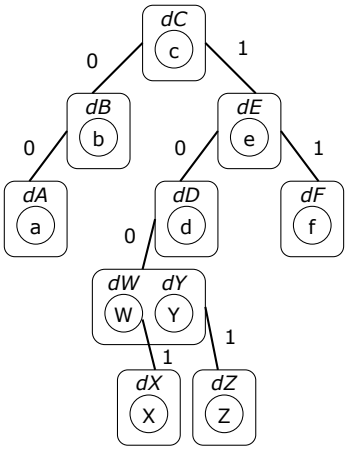

Figure 4. Identifiers after inserting atom between mini-siblings

$PosID_d$, thus $PosID_Y$ has no right child). If concurrently other user inserts atom `W` between `c` and `d`, it also creates a left child $[10(0 : dW)]$ under major node containing `d`. This leads to a major nodes containing two mini-nodes, as it is illustrated in Figure 3 (assuming $dW < dY$)

If, subsequently, some user wants to insert `X` between `W` and `Y`, this creates a child under mini-node `W`, where $id(X)$ =$[10(0 : dW)(1 : dX)]$ (rule in line 6, where PosID nodes are siblings). This is illustrated in Figure 4.

3.3. Disambiguators

Several alternatives are possible for disambiguators. Here, we discuss two.

3.3.1. Unique disambiguators (UDIS). A disambiguator may be represented as a $(counter, siteID)$ pair, where $siteID$ is a globally-unique site identifier (e.g., its MAC address), and $counter$ is a per-site persistent counter. Such disambiguators are ordered as follows: $(c_1, s_1) < (c_2, s_2)$, iff: $c_1 < c_2 \vee (c_1 = c_2 \wedge s_1 < s_2)$. This approach ensures that every disambiguator is unique; we note this approach UDIS.

With UDIS, a leaf mini-node can be discarded as soon as it is deleted. When a non-leaf mini-node is deleted, its atom may be discarded immediately, but the node itself must be kept. However, if all its descendants become deleted, recursively this node is discarded too. If all the mini-nodes of a major node are deleted, and all its descendants, then the major node is discarded. Conversely, the replay version of $insert$ may find that ancestors of the new node have been discarded concurrently, and must re-create empty nodes to replace them.

3.3.2. Site disambiguators (SDIS). A simpler alternative for disambiguators, noted SDIS, is to use a site identifier, with no counter.

Furthermore, we consider two alternatives for site identifiers. (1) MAC addresses can be used, as above. Or, (2) in a system with known membership, site identifiers can be more compact: each site is assigned a short integer for the duration of its membership.

SDIS has lower overhead than UDIS, but is not sufficient to avoid having two atoms with the same PosID. Consider for instance the following scenario. Site A inserts an atom with PosID a. After this, site B inserts an atom $a \odot 1$. Concurrently, site C deletes a; site A replays the delete. If node a is discarded, A could again insert an atom with PosID a.

To avoid this problem, a delete does not discard the node. It does discard the node's atom, and marks the node as a *tombstone*. Later in this paper, we will study approaches to garbage-collect such tombstones.

4. Optimizations

The approach presented so far has some limitations that we address in this section.

4.1. Keeping the tree balanced

The simple algorithm for generating new PosIDs for new atoms presented in section 3.2 makes no effort to keepi the tree balanced. In some cases, for instance if a user always appends to the end, the paths will grow with each new atom. To address this problem, the key observation is that, when generating a new PosID, it is not necessary to generate an

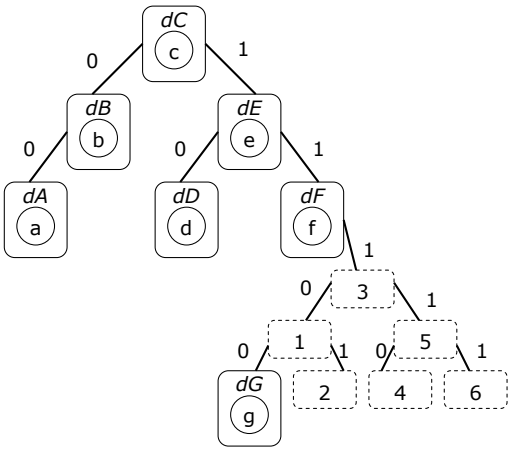

Figure 5. Identifiers for balanced tree

immediate descendant of an existing node. Thus, we propose an alternative heuristic that will grow the height, h, of the tree by $\lceil log_2(h) \rceil + 1$.[2] The new PosID will be the smallest of PosIDs in the grown tree. Thereafter, new PosIDs would be generated by using empty position in the tree of identifiers.

From the example in Figure 2, when inserting atom g in the end of the document, the tree would be grown by three levels, leading to the new PosID $[1110(0:d)]$. Thereafter, the following atoms would consecutively use the PosIDs for the empty nodes in the sub-tree, as numbered in Figure 5.

4.2. Minimizing structure overhead

The proposed approach imposes overhead for maintaining PosIDs for each atom. When implementing Treedoc as a list of $(atom, PosID)$ couples, for each atom it is necessary to maintain a $PosID$. When implementing Treedoc as a tree, the $PosID$ is the path in the tree being only necessary to store one disambiguator for each atom. However, keeping the tree structure imposes additional overhead that can be even more costly, depending on how the tree is maintained.

This problem can be mitigated relying on garbage-collection mechanisms. For instance, disambiguators could be removed once it is clear they are not necessary, i.e., that there is a single mini-node (note that sibling mini-node only occur as a consequence of concurrent inserts). Deleted nodes can be garbage-collected even when using site identifiers as soon as it is clear that every site has already deleted the atom and no operation refereing to it will be issued.

In this subsection, we propose a more radical solution: structural clean-up operations that switch between the efficient sequential buffer representation with no additional metadata, and the more expensive edit-oriented representation with PosIDs. The specification of these operations is as follows.

2. When inserting a known sequence of atoms, we could just create the smallest sub-tree that could store all new atoms.

Algorithm 2 *explode* and *flatten*

1: **procedure** $explode$ $(atomarray)$ // $atomarray$: sequence of atoms
2: $depth = \lceil \log_2(length(atomarray) + 1) \rceil$
3: T = Allocate a complete binary tree of depth $depth$
4: Assign identifiers for T nodes in infix order to the atoms of $atomarray$
5: Remove any remaining nodes
6: Return T

- $explode(atomarray)$. Returns a tree-storage Treedoc whose contents is identical to $atomarray$.
- $flatten(path)$ Returns an atom array whose contents is identical to the sub-tree rooted at $path$.

The initiator and replay versions of these operations must have identical effect. In particular, $explode$ must return exactly the same structure at all sites.

Observing that the capacity of a complete binary tree with $depth$ levels is $2^{depth} - 1$, we suggest for $explode$ the simple implementation of Algorithm 2. Observe that after $explode$, a path of an atom is a simple bitstring, with no disambiguators.

These internal clean-up operations do not genuinely commute with edit operations. We address this issue next.

4.2.1. Commuting clean-up with edits. A first observation is that the $explode$ operation is just a mapping of an array to a canonical tree representation. Array storage is converted to tree storage when necessary, e.g., when applying a path to an array. Therefore we can eliminate explicit $explode$ operations, and commutativity is not an issue.

A second observation is that *flatten* is not an essential operation. When *flatten* is concurrent with an edit operation in the same subtree, then the edit should have higher precedence. More precisely, a conflicting edit causes a *flatten* to abort, leaving no side-effects (and causing no harm).

Therefore *flatten* executes a distributed commitment procedure. When executing *flatten* at some site, if this site observes the execution of a $insert$, $delete$ or *flatten* within the sub-tree to be flattened, that site votes "No" to commitment, otherwise it votes "Yes." The operation succeeds only if all sites vote "Yes," otherwise it has no effect. Any distributed commitment protocol from the literature will do, for instance two-phase commit, three-phase commit, or Gray and Lamport's fault-tolerant protocol [6]. (We have designed a protocol that allows *flatten* to succeed even when some replicas are disconnected and to allow users to make contributions while disconnected, but this is out of scope of the current paper.)

We may now envisage a mixed tree, where parts that are currently being edited are in Treedoc representation, and parts that are currently quiescent are represented as arrays, with no associated metadata.

Document	Flatten	PosID Max	PosID Avg	Nodes Number	Nodes bytes	Nodes Mem ovhd	Nodes % non-Tomb	On-disk overhead bytes	On-disk overhead % doc
Distributed Computing	no	237	76.09	2766	72,916	3.70	6.18	2405	12.21
(wiki, 171 paras,	1	167	27.73	816	21,216	1.08	20.96		
19,686 bytes, 870 revisions)	2	236	74.48	2757	71,682	3.64	6.20		
IBM POWER	no	190	82.52	1112	28,912	1.17	16.55	914	3.71
(wiki, 184 paras	1	14	9.32	338	8,788	0.36	54.44		
24,651 bytes, 401 revisions)	2	190	82.52	1112	28,912	1.17	16.55		
Grey Owl	no	113	51.61	866	22,516	1.82	12.70	760	6.13
(wiki, 110 paras,	1	86	32.20	416	10,816	0.87	26.44		
12,388 bytes, 242 revisions)	2	108	50.46	798	20,748	1.67	13.78		
acf.tex	no	170	66.10	1853	48,178	3.43	17.92	1527	10.86
(latex, 332 lines,	2	20	7.86	461	11,986	0.85	72.02		
14,048 bytes, 51 revisions)	8	170	67.70	1800	46,800	3.33	18.44		
algorithms.tex	no	91	27.98	1973	51,298	3.38	20.07	1581	10.41
(latex, 396 lines,	2	20	7.87	567	14,742	0.97	72.02		
15,186 bytes, 58 revisions)	8	170	67.70	1783	46,358	3.05	18.44		
propagation.tex	no	245	72.11	2484	64,584	2.91	19.36	1917	8.64
(latex, 481 lines,	2	12	7.39	495	12,870	0.58	97.17		
22,170 bytes, 68 revisions)	8	245	90.19	1538	39,988	1.80	31.27		

Table 1. Measurements. Document: name, type, size in paragraphs or lines, size in bytes, revisions. Flatten: no flattening, or number of revisions between flatten heuristics. PosID: maximum and average PosID length (in bits). Nodes: number, memory occupied (in bytes), overhead relative to document size, percentage of non-tombstone. On-disk overhead: absolute, relative to document size. The numbers relate to the final state of each document (i.e., after all revisions are applied). Empty cells were not measured.

5. Evaluation

In this section, we evaluate experimentally the behaviour of Treedoc. Our goal is to measure the overheads of Treedoc, and to evaluate the effects of some design alternatives: granularity of atoms, disambiguator design (UDIS vs. SDIS), and flatten heuristics.

To ensure a realistic evaluation, we replay co-operative edit sessions extracted from existing repositories. We analysed a large number of different workloads: edits of C++ source files from the open-source KDE project SVN repository; of Java and Latex source files from a private SVN repository; and of Wikipedia pages. However, we only present a small number of representative results. The characteristics of the documents studied here are summarised in Table 2.

Our experiments start by creating an initial Treedoc document containing the text of the initial version in the version control repository. Thereafter, for each revision, we compute the differences from the previous version, and execute an equivalent sequence of insert and delete operations to the Treedoc document.

Modifying an atom is modeled as deleting the original and inserting the modified atom. This results in an unexpectedly large number of deletes. This effect is especially noticeable for Wikipedia documents, since the atom granularity is large. Furthermore Wikipedia documents suffer vandalism episodes, in which large portions of text are repeatedly defaced, then restored by an administrator. This further increases the amount of deleted information.

Table 2. Summary of documents studied

	Number of revisions	Number of lines initial	Number of lines final
average	312	103	279
less active	51	99	332
most active	870	9	171

Table 3. Fraction of tombstones (LaTeX documents)

	no balancing	balancing
no-flatten	77.5%	77.5%
flatten-8	67.8%	62.9%
flatten-2	15.8%	14.2%

We use 6 bytes for site identifiers in both UDIS and SDIS, and 4 bytes for the UDIS counter. The granularity of an atom for Latex, C++ and Java files is a text line, usually under 80 characters; the Wikipedia granularity is a whole paragraph.

5.1. Garbage-collection and balancing efficiency

Our implementation of flatten heuristically attempts to identify "cold" areas of the document, i.e., subtrees that have not been recently modified. We tested different scenarios: without flattening, or selecting flattening some cold area every 1, 2 or 8 revisions. We evaluate its effectiveness in the SDIS variant. The "% non-Tomb" column of Table 1 shows the fraction of non-tombstones over the total nodes. When flattening is not used, up to 95% of nodes are tomb-

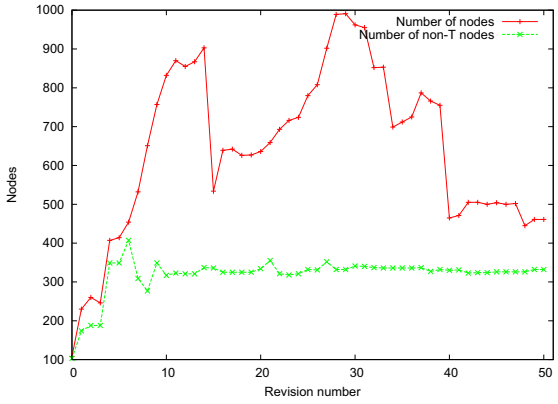

Figure 6. Variation of number of nodes for acf.tex

Table 4. SDIS vs. UDIS (LaTeX documents)

		no balancing		balancing	
		SDIS	UDIS	SDIS	UDIS
no-flatten	overhead/atom	570	140	377	107
	avg PosID size	108	140	74	107
flatten-8	overhead/atom	459	121	197	67
	avg PosID size	103	121	55	67
flatten-2	overhead/atom	34	24	32	25
	avg PosID size	27	24	24	25

stones. Flattening effectively garbage-collects tombstones, and this process generally improves when flattening more aggressively.

Figure 6 shows the evolution of the total number of nodes, as well as the number of non-tombstone nodes, over the lifetime of an example document (acf.tex in this case). Flattening appears as drastic reduction to the total number of nodes (upper curve). In some cases flattening succeeds in reducing the total number of nodes by 50%. However, this is lower than our expectations; we believe the heuristic choice of the sub-tree to flatten is to blame.

We also study a variant of the balancing strategy of Section 4.1, where we group all the consecutive inserts of a given revision into a minimal sub-tree. Table 3 shows some results combining our balancing heuristic, coupled with flattening, for LaTeX documents (results are similar for the other types of documents). It shows that balancing is effective and augments the effect of flattening. It confirms that, in the common case, it is best to flatten aggressively.

5.2. Costs

We distinguish several costs: CPU, in-memory storage, on-disk storage, and network. While we have not measured CPU time precisely, we know it to be negligible in the absence of flattening, as our simulations run very quickly, e.g., less than 1.44 seconds for the "Distributed Computing" Wikipedia entry.

When maintaining the document as a sequence of $(atom, PosID)$ couples, in-memory overhead is equal to the total size of identifiers of all nodes, including tombstones (only when SDIS are used). In this case, the in-memory overhead is the number of Treedoc nodes multiplied by the average size of an PosID.

When maintaining the document in a tree structure, in-memory overhead is equal to the total size of Treedoc nodes, including tombstones. A standard node includes the number of non-tombstone nodes in its subtree, pointers to its left and right children, a disambiguator, and a pointer to the atom, which comes to 26 bytes (plus Java overhead) on a 32-bit machine. Various optimisations are possible, e.g., a flattened node contains an array of atom pointers; we do not consider these optimisations hereafter. A node with mini-nodes replaces the disambiguator by an array of $\{$*node, disambiguator*$\}$ pairs, but this case does not occur in our tests, since SVN and Wikipedia serialise their edits. Thus an upper bound on in-memory overhead is the number of Treedoc nodes without flattening, multiplied by 26 bytes. Table 1, under "Nodes/Mem ovhd," shows that the in-memory overhead remains reasonable (between 0.36 and 3.7 times the file size). Figure 6 shows the variation over the lifetime of document acf.tex. It is apparent that our current flattening heuristics are not effective; flattening the final version would bring overhead down to zero.

To compare the SDIS vs. UDIS approach to disambiguators, refer to Table 4. It shows that, although the overhead of UDIS is larger per node, the total overhead is lower. This is because UDIS lowers the number of nodes significantly, by eliminating tombstones early. Therefore, the UDIS approach is better in the common case.

In order to store a Treedoc on disk, we use a modified version of the well-known technique that represents a binary heap of depth i as an array of size 2^i. Nodes are stored from top to bottom, line by line, and nodes on the same line are stored left to right. Each array entry contains a disambiguator and a reference to the corresponding atom (stored in a separate file). For every node that has only a single descendant or no descendants, we fill the places with a special marker. To save space, we compress sequences of markers with run-length encoding. Table 1, column "On-disk overhead" measures the on-disk size of the final revision of several documents.

We cannot yet evaluate the cost of a distributed flatten, as it is not currently implemented. The network cost of an edit operation is sending a PosID and, when inserting, the corresponding atom. An upper bound on the total network cost of building a document from scratch can be estimated by adding the size of PosIDs for all atoms. An estimate of average network cost is the average PosID size, found in Table 1, under "PosID/Avg."

Table 5. Comparing Treedoc vs. Logoot: PosID sizes

	PosID size ratio (Logoot/Treedoc)
Distributed Computing	2.4
IBM POWER	1.8
Grey Owl	1.9
acf.tex	3.2
algorithms.tex	3.9
propagation.tex	3.6

5.3. Comparison with Logoot

We offer a brief comparison with Logoot, a recent CRDT for co-operative editing [7]. A Logoot position identifier is a sequence of fixed-sized unique identifiers. Position identifiers are ordered in lexicographical order of their components. Logoot allocates position identifiers sparsely in order to facilitate insertions. To insert, Logoot allocates a free unique identifier ordered between the left and right position identifiers, if one exists; otherwise it extends the identifier of the left position with an additional layer. A deleted atom can be removed immediately, but Logoot does not flatten the tree. Even without flattening, we expect Logoot to have larger overhead than Treedoc, since its position identifiers are larger than Treedoc's.

This is confirmed by our measurements in Table 5, comparing the total PosID size of Logoot and Treedoc/ UDIS, without flattening. We use the same size for UDIS and Logoot unique identifiers (10 bytes). With flattening, the comparison would be even stronger in favour of Treedoc.

6. Related work

A comparison of several approaches to the problem of collaboratively editing a shared text was written by Ignat et al. [8].

Operational transformation (OT) [2, 9–12] considers collaborative editing based on non-commutative operations. To this end, OT transforms the arguments of remote operations to take into account the effects of concurrent executions. OT requires two correctness conditions [13]: the transformation should enable concurrent operations to execute in either order, and furthermore, transformation functions themselves must commute. The former is relatively easy. The latter is more complex, and Oster et al. [3] prove that all previously proposed transformations violate it. Later solutions [14–16] are complex, and their correctness is hard to verify.

OT attempts to make non-commuting operations commute after the fact. We believe that a better approach is to design operations to commute in the first place. This is more elegant, and avoids the complexities of OT.

A number of papers study the advantages of commutativity for concurrency and consistency control [17, 18, for instance]. Systems such as Psync [19], Generalized Paxos [20], Generic Broadcast [21] and IceCube [22] make use of commutativity information to relax consistency or scheduling requirements. However, these works do not address the issue of achieving commutativity.

Weihl [18] distinguishes between forward and backward commutativity. They differ only when operations fail their pre-condition. In this work, we consider only operations that succeed at the submission site, and ensure by design that they won't fail at replay sites.

Roh et al. [23] independently proposed the CRDT approach. They give the example of an array with a slot assignment operation. To make concurrent assignments commute, they propose a deterministic procedure (based on vector clocks) whereby one takes precedence over the other. This approach (similar to the Last-Writer Wins algorithm of shared file systems) is destructive and loses work. Roh does not consider the case of co-operative editing, where concurrent updates should be merged, not lost.

Oster et al. propose the WOOT algorithm for managing cooperative editing, supporting insert and delete operations [24]. Although not identified as such by the authors, WOOT is also a CRDT. In WOOT, each character has a unique identifier, and maintains the identifiers of the previous and following characters at the initial execution time. Furthermore, the data structure grows indefinitely, because there is no garbage collection or restructuring.

More recently, Weiss et al. proposed the Logoot CRDT [7]. Logoot uses a sparse n-ary tree rather than Treedoc's dense binary tree. A position identifier is a list of (long) unique identifiers, and Logoot does not flatten. As we show above, Logoot has a high overhead compared to Treedoc.

An alternative approach to consistency is executing operations in the same order at all replicas [1]. However, to ensure that an edit position has the same meaning at all replicas requires either operating replicas in lock-step, or operational transformation [2]. In the Treedoc design, common edit operations execute optimistically, with no latency; replicas synchronise only in the background.

7. Conclusions

It was known previously that commutativity simplifies consistency maintenance, but the issue of designing systems for commutativity was neglected. This paper suggested a new paradigm for replication: the Commutative Replicated Data Type or CRDT, designed such that concurrent operations commute. If operations replay in happened-before order, replicas of a CRDT converge automatically, without complex concurrency control. This makes the implementation of replicated systems much simpler than before.

However, designing a CRDT with the desirable property that no work is lost (genuine commutativity) is not easy. We have presented Treedoc, a genuine CRDT solution to the

problem of a shared edit buffer, that relies on an extended binary tree as the base for building unique identifier with the needed properties.

We have proposed two identification alternatives: one is compact but uses tombstones to keep track of deleted entries; the other allows deleted entries to be discarded immediately. As a tree that is badly unbalanced causes overhead, we suggest optimizations to avoid unbalance.

We performed a performance analysis using traces from existing editing histories. It shows that Treedoc has smaller overhead than the Logoot alternative, and a reasonable one in absolute terms.

Our next step in this research will be to enable peer-to-peer co-operative editing at a large scale, by implementing Treedoc within an existing text editor or wiki system.

Acknowledgment

We thank Pascal Urso and his co-authors for providing us with an early version of their Logoot paper [7]. This work is supported in part by the Portuguese FCT/MCTES project POSC/59064/2004, with FEDER funding, the EU FP6 project Grid4All, the French ARA project Respire, the French ARC project Recall, and the Spanish TIN2008-01288 project.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[2] C. Sun and C. Ellis, "Operational transformation in real-time group editors: issues, algorithms, and achievements," in *Int. Conf. on Computer-Supported Coop. Work (CSCW)*, Seattle WA, USA, Nov. 1998, p. 59.

[3] G. Oster, P. Urso, P. Molli, and A. Imine, "Proving correctness of transformation functions in collaborative editing systems," LORIA – INRIA Lorraine, Rapport de recherche RR-5795, Dec. 2005. [Online]. Available: http://hal.inria.fr/inria-00071213/

[4] M. Shapiro and N. Preguiça, "Designing a commutative replicated data type," Institut National de la Recherche en Informatique et Automatique, Rocquencourt, France, Rapport de recherche RR-6320, Oct. 2007. [Online]. Available: http://hal.inria.fr/inria-00177693/

[5] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai, "Transparent adaptation of single-user applications for multi-user real-time collaboration," *ACM Trans. on Computer-Human Interaction (TOCHI)*, vol. 13, no. 4, pp. 531–582, dec 2006.

[6] J. Gray and L. Lamport, "Consensus on transaction commit," *Trans. on Database Systems*, vol. 31, no. 1, pp. 133–160, Mar. 2006.

[7] S. Weiss, P. Urso, and P. Molli, "Logoot: a scalable optimistic replication algorithm for collaborative editing on P2P networks," in *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, Montréal, Canada, Jun. 2009.

[8] C.-L. Ignat, G. Oster, P. Molli, M. Cart, J. Ferrié, A.-M. Kermarrec, P. Sutra, M. Shapiro, L. Benmouffok, J.-M. Busca, and R. Guerraoui, "A comparison of optimistic approaches to collaborative editing of Wiki pages," in *Int. Conf. on Coll. Computing: Networking, Apps. and Worksharing (CollaborateCom)*, no. 3, White Plains, NY, USA, Nov. 2007.

[9] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Int. Conf. on the Mgt. of Data (SIGMOD)*, ACM SIGMOD. Portland, OR, USA: ACM, 1989, pp. 399–407.

[10] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *Trans. on Comp.-Human Interaction*, vol. 5, no. 1, pp. 63–108, Mar. 1998.

[11] D. Li and R. Li, "Preserving operation effects relation in group editors," in *Int. Conf. on Computer-Supported Coop. Work (CSCW)*. New York, NY, USA: ACM, 2004, pp. 457–466.

[12] ——, "Ensuring content and intention consistency in real-time group editors," in *Int. Conf. on Distributed Comp. Sys. (ICDCS)*. Hachioji, Tokyo, Japan: IEEE Computer Society, Mar. 2004, pp. 748–755.

[13] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser, "An integrating, transformation-oriented approach to concurrency control and undo in group editors," in *Int. Conf. on Computer-Supported Coop. Work (CSCW)*. Boston, MA, USA: ACM, May 1996, pp. 288–297.

[14] G. Oster, P. Molli, P. Urso, and A. Imine, "Tombstone transformation functions for ensuring consistency in collaborative editing systems," in *Int. Conf. on Coll. Computing: Networking, Apps. and Worksharing (CollaborateCom)*. Atlanta, Georgia, USA: IEEE Computer Society, Nov. 2006, p. 10.

[15] D. Sun and C. Sun, "Operation context and context-based operational transformation," in *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. New York, NY, USA: ACM, 2006, pp. 279–288.

[16] R. Li and D. Li, "A new operational transformation framework for real-time group editors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 3, pp. 307–319, 2007.

[17] B. R. Badrinath and K. Ramamritham, "Semantics-based concurrency control: beyond commutativity," *Trans. on Database Systems*, vol. 17, no. 1, pp. 163–199, Mar. 1992.

[18] W. E. Weihl, "Commutativity-based concurrency control for abstract data types," *IEEE Trans. on Computers*, vol. 37, no. 12, pp. 1488–1505, Dec. 1988. [Online]. Available: http://www.computer.org/tc/tc1988/t1488abs.htm

[19] S. Mishra, L. Peterson, and R. Schlichting, "Implementing fault-tolerant replicated objects using Psync," in *Symp. on Reliable Dist. Sys.* Seattle, WA, USA: IEEE, Oct. 1989, pp. 42–52. [Online]. Available: http://ieeexplore.ieee.org/iel2/259/2469/00072747.pdf?tp=&isnumber=2469&arnumber=72747

[20] L. Lamport, "Generalized consensus and Paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, Mar. 2005. [Online]. Available: ftp://ftp.research.microsoft.com/pub/tr/TR-2005-33.pdf

[21] F. Pedone and A. Schiper, "Handling message semantics with generic broadcast protocols," *Distributed Computing Journal*, vol. 15, no. 2, pp. 97–107, 2002. [Online]. Available: http://www.inf.unisi.ch/faculty/pedone/papers/2002DC.pdf

[22] N. Preguiça, M. Shapiro, and C. Matheson, "Semantics-based reconciliation for collaborative and mobile environments," in *Int. Conf. on Coop. Info. Sys. (CoopIS)*, ser. Lecture Notes in Comp. Sc., vol. 2888. Catania, Sicily, Italy: Springer-Verlag GmbH, Nov. 2003, pp. 38–55. [Online]. Available: http://www-sor.inria.fr/~shapiro/papers/coopis-2003.pdf

[23] H.-G. Roh, J.-S. Kim, and J. Lee, "How to design optimistic operations for peer-to-peer replication," in *Int. Conf. on Computer Sc. and Informatics (JCIS/CSI)*, Kaohsiung, Taiwan, Oct. 2006. [Online]. Available: http://kernel.kaist.ac.kr/~jinsoo/publication/csi06.pdf

[24] G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for P2P collaborative editing," in *Int. Conf. on Computer-Supported Coop. Work (CSCW)*. Banff, Alberta, Canada: ACM Press, Nov. 2006, pp. 259–268.